(2)

September 1989

UILU-ENG-89-2233
CSG-110

COORDINATED SCIENCE LABORATORY
College of Engineering

PATH ANALYSIS: A METHOD FOR ANALYZING MESSAGE COMMUNICATION IN FAULTY HYPERCUBES

Michael Paul Peercy

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-89-2233 (CSG 110)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois — 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 800 N Quincy Arlington VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research — 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N 00014-88-K-0624
8c. ADDRESS (City, State, and ZIP Code) See 7b	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

PATH ANALYSIS: A METHOD FOR ANALYZING MESSAGE COMMUNICATION IN FAULTY HYPERCUBES

12. PERSONAL AUTHOR(S)
PEERCY, Michael Paul

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ____ TO ____	1989 September 22	60

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Circuit Switching and Packet-Switching Delay, Hypercube Multiprocessor, Probabilistic Distributions, Reconfig. Strat.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

In this paper we present an analysis of communication in a hypercube multiprocessor which can be applied to systems in various degraded conditions under failure. By characterizing the links of the hypercube as a collection of overlapping paths, we develop Path Analysis, a method to calculate traffic and delay on each individual link in either a packet-switched or circuit-switched hypercube network. We can thus examine the degradation from ideal communication performance due to different fault patterns and different reroute strategies we may use to compensate for faults. Path Analysis also can account for the redistribution of tasks from faulty processors.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
[✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

PATH ANALYSIS: A METHOD FOR ANALYZING MESSAGE COMMUNICATION IN FAULTY HYPERCUBES

BY

MICHAEL PAUL PEERCY

B.S., Purdue University, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1989

Urbana, Illinois

ABSTRACT

This thesis presents an analysis of communication in a hypercube multiprocessor which can be applied to systems in various degraded conditions under failure. By characterizing the links of the hypercube as a collection of overlapping paths, Path Analysis is developed. Path Analysis is a method to calculate traffic and delay on each individual link in either a packet-switched or circuit-switched hypercube network. One can thus examine the degradation from ideal communication performance due to different fault patterns and different reroute strategies which may be used to compensate for faults. Path Analysis also can account for the redistribution of tasks from faulty processors.

ACKNOWLEDGEMENTS

I wish to thank Professor Prith Banerjee for his ideas, leadership, and assistance in the production of this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1.

INTRODUCTION

Recent advances in technology have allowed the development of message-passing multiprocessors with large numbers of processors that are directly connected through various topologies such as hypercubes, trees, and meshes [1,2]. These network systems generally have a routing scheme dictating what path a message takes from its source, through intermediate nodes, to its destination (or sink). We assume a deterministic routing scheme; that is, for each source node and each sink node in the given topology, there is a unique path which every message must take from that source to that sink.

Owing to the increased complexity of these multiprocessor systems, there is a large probability of failures of one or more processors or links in the system [3]. Hence researchers have worked on various fault tolerant network architectures [4,5].

We will call a fault a *processor fault* if it affects only the processing element of a node without damaging any communication paths through the node. A *node fault* is one which destroys communication through the node as well. A *link fault* does not harm the processor on either end of it; it only renders the link between them useless. A fault destroys the paths which use the faulty component; that is, the sources of those paths cannot send messages to their respective destinations. A reroute strategy may be used to make a new path around a faulty component.

In this thesis we propose a powerful method to analyze various types of networks under faults. Although these means may be used to analyze, with more or less difficulty, a number of different network topologies, we concentrate on the hypercube interconnect topology with circuit-switched communication.

The first generation hypercubes used packet-switching communication [6, 7, 8, 9]. Current generation hypercubes use circuit-switching in an attempt to decrease the latency of message transmission [10, 11]. The analysis of traffic in a hypercube network does not depend on the type of switching (packet or circuit). However, the formulation of accurate models for delay is much more difficult in the case of circuit-switching, and although packet-switched networks are thoroughly analyzed in the literature, there seems to be no prior analysis of delay in circuit-switched computer networks.

We propose a novel and potent method for characterizing the message traffic on each link of the network. We show the application of the traffic thus found to past models of packet-switched circuits in order to find delay, and we use the traffic in an original analysis of delay assuming the network is circuit-switched.

The method we propose is called Path Analysis (PA). Essentially PA consists of representing the hypercube topology as a collection of paths. These paths physically coincide on the links of the hypercube. By characterizing, as generally as necessary, the paths on a link l, we can describe the traffic on l as the sum total effect from all of the coincident paths. The traffic, or throughput, is how many messages must be transmitted across link l. When the effects of waiting for l and other links are taken into account, the delay of a message on l can be described.

The thesis is presented as follows. Chapter 2 discusses related work in reconfiguration and reroute strategies. Chapter 3 is a theoretical presentation of Path

Analysis. Chapter 4 describes a program written to exploit PA with ease and gives a comparison between the PA algorithm's estimates of traffic and delay and results drawn from simulation. Chapter 5 obtains results from PA program runs. Chapter 6 draws conclusions and presents extensions on the Path Analysis idea.

CHAPTER 2.

REVIEW OF RELATED WORK

A symmetric network is one which looks identical to every node in the network. In symmetric networks we can see each link as identical, and the traffic we calculate for that one link applies to all links. A fault-free hypercube is clearly a symmetric network, but any fault in the hypercube links, processors, or nodes makes the network asymmetric. Because of the difficulty of describing asymmetric networks, most analyses are of symmetric networks. Reed and Fujimoto [2] give one such analysis. From this work, we use notation and vocabulary for our analysis.

Once a fault has occurred, we would like to be able to bypass the fault with minimal degradation of performance. The value of a reconfiguration strategy is measured by how little the performance is degraded. Our analysis is developed as a tool to measure the effects of faults and reconfiguration strategies on optimum performance of the hypercube.

Two issues that need to be addressed in faulty multiprocessors are (1) reconfiguration, i.e., process remapping on nodes, and (2) rerouting of messages. We now review some related work on both issues.

2.1. Review of Reconfiguration Strategies

If the fault in the hypercube is a link fault, no process remapping is required. However, if it is a node fault or processor fault, there is a need for reconfiguration. Several strategies have been proposed.

One approach involves the use of spare nodes and links such that under failure the faulty nodes and links are replaced by spares [12, 13, 14]. After reconfiguration, another fault-free hypercube is obtained, and the analysis of this reconfiguration strategy becomes trivial.

A second approach to reconfiguration after failure of some nodes involves the determination of the largest fault-free subcube in the faulty hypercube, and running the application on it [15, 16, 17, 18]. The disadvantage of this approach is the waste of possibly good processors. Again since the resultant network is still a hypercube (a symmetric network), the analysis is straightforward.

A third approach is the load redistribution of the work on the faulty processors to the remaining fault-free processors. Numerous papers have addressed this problem in the domain of loosely coupled distributed systems [19, 20, 21, 22, 23]. Specifically, for hypercubes, some theoretical work has been proposed in [24]. On a practical side, the MOOSE operating system at Caltech [25] uses an object-oriented model to perform task redistribution. Recently, Banerjee [26] proposed a software scheme using virtual processors to perform load distribution to n fault-free neighbors of a faulty node. This strategy is quite general and can encompass the other strategies as well. We will use this strategy to illustrate the capability of our Path Analysis method later in this thesis.

2.2. Review of Reroute Strategies

Reroute strategies are designed to send messages around faults. They become very important with respect to link faults, which destroy communication lines but not any computational power. Also, after task redistribution due to a node fault, some useful form of routing around the faulty node is necessary if the degraded hypercube is to meet its highest possible performance. Numerous proposals and investigations have been made regarding routing and broadcasting in faulty hypercubes [27,28,29]. Routing schemes that are designed to avoid network congestion also can provide fault tolerant rerouting [11].

If normal communication can not proceed due to a fault, the reroute algorithms determine how messages should be rerouted around the fault. We will describe three varieties of reroute strategies: adaptive, single-alternate, and table. We call a reroute *adaptive* if it sends a message along a path, altering the path as it finds faults in it. We developed the *single-alternate* strategy as follows: if a message fails to be routed successfully by the standard routing strategy, the source sends the message from source to sink along a single alternative path. *Table routing* strategies use an occasionally executed algorithm to determine the best path between each source and sink, storing these in a lookup table at every node.

First, we present the standard routing procedure which is the default of all our discussed reroutes when the cube is fault-free. This routing is called *ecube* [30], the most popular routing strategy in hypercube systems today. In *ecube* the bit differences between the source and destination of a path are resolved from low dimension to high.

Ecube ensures the shortest path in a perfect cube and avoids deadlock. A problem with *ecube* which we explore later is that, in a circuit-switched hypercube network, lower dimensional links must wait for higher dimensional links to free up in order to make long paths. Although all links may have the same traffic on them, lower links have greater delay.

We examine two adaptive routing strategies. Chen and Shin [27] present a routing algorithm which we call *C&S adaptive* later in this thesis. In this algorithm each message carries a header which contains an ordered list of dimensions to be traversed and a n-bit tag. The list is initialized to the order the dimensions would be traversed in the *ecube* routing. At each node in the path, the links are tried in the order given in the list until an unfaulty one is found. The good link is then taken and that dimension is struck from the list. If all dimensions in the list put the path on faulty links, then a spare dimension is chosen from outside the list. The link in that dimension is taken, and the list is appeı ded by adding that dimension onto the end. The tag is used to mark dimensions derived from entirely blocked lists or taken as spares so that a bouncing effect does not take place.

The *hyperswitch* routing strategy, developed by JPL for its hypercube, is designed to avoid congestion as well as faults [11]. A message in the hyperswitch network uses a channel history tag (CHT) to mark dimensions it has found to be unavailable (busy or faulty). A message attempts to run along its *ecube*-determined path. If it reaches an unavailable link, it tries another link, backtracking if necessary, and modifies the CHT accordingly. The result is that a subset of the possible minimum-length paths from source to destination is tried quickly and efficiently. It must be stressed that the hyperswitch network is designed to speed the routing of messages by avoiding the wait on congested links; its primary purpose is not fault-tolerance, although it succeeds in being a viable reroute method.

The single-alternate reroute strategies are worth examining because of their acceptable handling of single faults in the hypercube and their simple implementation. In these methods the *ecube* path is tried, and, if a fault is found, one alternate path is tried. The single-alternate path should have the property of being completely disjoint from the *ecube* path. If the alternate path fails, the message routing is unsuccessful. Two single alternate reroutes we developed are *high dimension first* (*highfirst*) and *reverse ecube* (*reverse*).

If the list of the dimensions to be taken in an *ecube* route are cyclically rotated, a disjoint path is created (assuming more than one dimension in the list) [28]. *Highfirst* selects an alternative path by first using the highest dimension bit differing source and sink; the remaining dimensions are traversed in their low to high *ecube* order. The ease of implementation of this method is clear when we recognize that only the source needs to know that a reroute has occurred. The source tries the *ecube* path. If that path fails, this fact is returned to the source. The source then sends the message to its highest dimension neighbor along the dimensions which need to be traversed. That neighbor then proceeds as though it were transmitting a standard *ecube*-routed message.

Another reroute method we have developed is called *reverse*. In the context of the single-alternate reroute, *reverse* can be described as follows. If the *ecube* path fails, we can try to make a path by traversing the dimensions from high to low, exactly in reverse of our definition of *ecube*. This is easy to implement because a single bit can be carried in the message header to tell whether the message is proceeding forward or backward through the dimensions. Each node along the path merely examines that bit to determine whether the message should exit along the lowest unresolved dimension or the highest.

The third type of reroute strategy we examine is table routing. In table routing each node's routing processor holds a table indexed by message destination address. Each table location contains the dimension which should be traversed from the current node in order to reach the destination. Some algorithm must be run which fills the table in each node with the optimum set of paths for the hypercube.

Banerjee [26] developed an optimal table-filling algorithm which is run in a centralized fashion, i.e., by the host. It is modeled from Dijkstra's algorithm for minimal paths in general graphs. From this algorithm we proposed a distributed algorithm which could be run by the processors of the hypercube without the host's intervention. Although both the centralized and distributed table-filling algorithms find shortest paths, they do not necessarily find the same path. The centralized version builds paths from source to destination while the distributed version builds paths from destination to source. We therefore consider these algorithms separately in our investigations. Irrespective of how the tables are filled, whether by centralized or distributed system programs, the actual routing becomes trivial. Any node that originates or passes along a message just checks the destination address, looks up the output link, and sends the message out that link. The random access memory (RAM) that is necessary for this table is approximately N by $logN$ in size, where N is the number of processors in the hypercube. Another bit per word should be used to mark nodes as inaccessible, faulty, or reached. Another advantage besides simplifying routing logic to table lookup is that a source knows immediately whether or not a destination can be reached (is accessible) or should be reached (is not faulty) from its table. If we reserve a bit or bit pattern to identify the current node, a message can know immediately whether or not it has reached its destination from the RAM lookup.

We have introduced a number of reroute methods of different types. We cannot, however, analyze these methods with the techniques given in works on symmetric or fault-free networks. Once a fault has been introduced, an asymmetry is produced. Reroute strategies further complicate the matter by placing traffic which once used faulty components onto unfaulty ones. Yet we wish to compare these methods in an analytical and fast way to determine which is best for our needs and our prediction of faults. We propose the idea of Path Analysis to do this evaluation, which is the key contribution of this thesis.

CHAPTER 3.

THEORETICAL FRAMEWORK FOR PATH ANALYSIS

Through our discussion of reroute techniques, we have somewhat motivated the need for analysis methods for these techniques. Essentially we wish to examine absolute and relative densities of traffic and lengths of delays on the various links of a degraded hypercube. The tool we develop here to make this examination is Path Analysis. In PA we choose any link l, we find all paths which use link l, and then we quantitatively describe the total effect of these paths to give traffic. The traffic on l, given a little more description of the paths on l and its adjacent links, can yield an approximation for the delay on l.

3.1. Notations

First we should introduce our conventions and notations. Our hypercube is of dimension n. Its nodes are addressed with the appropriate n-bit number, the links connecting nodes which differ in exactly one bit. The dimension of a link is the bit which differs between its endpoints. We will use the terms lower link and higher link to mean one of lower or higher dimension. It is important, but not necessary, in our analysis to view neighboring nodes as connected by two links, not only one. In recent hypercubes [10] communication can occur both ways, that is, communication is bidirectional, on one set of connections. As a result, we view this single set of connections as two links. We should note that a fault anywhere within this set of connections will generally make both links inoperable due to the interdependence of the two links' traffic.

As a result of seeing two links for every logical connection, we have $M = n2^n$ links total. We address a link with the node it exits, indicating with a hat the dimension of the link, that is, the bit which is changed in traversing the link. For example, we give a four-dimensional hypercube in Figure 1. In this figure, label A identifies link $001\hat{1}$ which traverses the 0^{th} dimension between nodes 0011 and 0010. Link $001\hat{0}$ crosses the same gap, but in the opposite direction. Both links $001\hat{1}$ and $001\hat{0}$ can be used at the same time, and are considered separately in our analysis. As another example of link addresses we label link $\hat{0}000$ as B.

We represent a path with the notation $(s\ d)$, where s is the source of the path and d is the destination. Notice that all the routing strategies we have discussed are entirely deterministic, that is, given a particular degraded condition of the hypercube F, there is at most one successfully taken order of links from s to d. This property is critical in PA. Therefore, for a particular F, $(s\ d)$ is equivalent to an ordered list of links $[l_1\ l_2\ ...\ l_k]$, determined by F and the routing methods. For example, in Figure 1 *ecube* would route (0011, 1000) from the lowest dimension to the highest as $[001\hat{1}, 00\hat{1}0, \hat{0}000]$.

We need to quantitatively model the paths on the links as numbers, simplifying them in terms of how often they are used and how long they are used. Although more general descriptions are possible, we choose to use the destination distribution model given in [2] to represent the frequency of use of a path. Reed and Fujimoto define the *destination distance distribution* $\Phi(k)$ as the probability that a message crosses k links. They also define $Reach(k)$ to be the number of destination nodes exactly k links away from an arbitrary source in a symmetric network. We change the definition of $\Phi(k)$ slightly by dividing $\Phi(k)$ by $Reach(k)$. The resulting *destination distribution* ψ_k is the

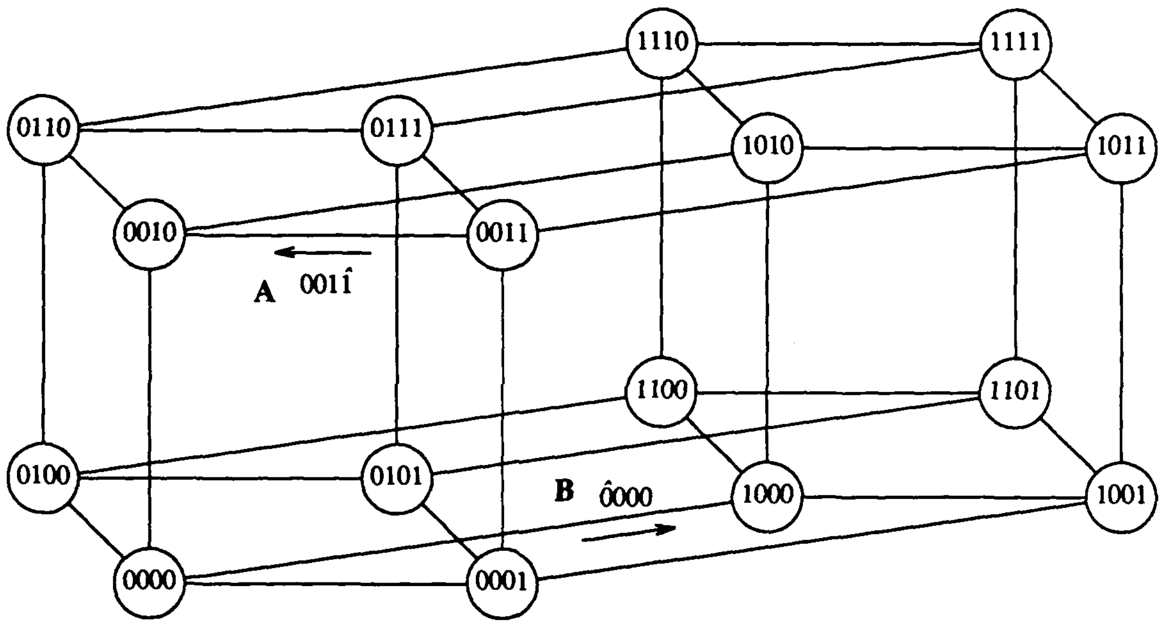


Figure 1. Four-Dimensional Hypercube (16 processing nodes)

probability that an arbitrary message is going to a chosen destination distance k from the source. The n-vector of these probabilities we call ψ.

Useful examples from [2] describe $\Phi(k)$ and ψ. Two types of destination distance distributions are uniform and sphere of locality. The *uniform* distribution is the one in which every destination has the same probability of receiving an arbitrary message. In terms of $\Phi(k)$, $\Phi(k)=Reach(k)/(2^n-1)$; in terms of ψ, $\psi_k=1/(2^n-1)$. A *sphere of locality* distribution defines a distance r; some destination within r links of the source receives a message with probability ϕ, and some destination farther than r links from the source receives a message with the complementary probability $1-\phi$. Here ϕ is the probability that a message is sent to a destination no more than distance r from the source, the destination distance distribution $\Phi(k)$ is the probability that a message is sent to a destination exactly distance k from the source, and the destination distribution ψ_k is the probability that a message is sent to a *particular* destination which happens to be distance k from the source.

To further illustrate the difference between $\Phi(k)$ and ψ_k, assume the four-dimensional hypercube in Figure 1. ψ_k, from the formula given above, is 1/15. This is the probability that the source sends the message to each other node. In other words, if node 0 sources a message, there is a 1/15 probability that the message is to node 2; there is also a 1/15 probability that the message is to node 14; and so on. $\Phi(k)$ classifies all nodes a given distance k in one number, assuming each node in that class is selected uniformly once the class is selected. $\Phi(1)$ is 4/15, $\Phi(2)$ is 6/15, $\Phi(3)$ is 4/15, and $\Phi(4)$, the probability of sending a message from node 0 to 15, is 1/15.

The reason we use ψ rather than $\Phi(k)$ in our analysis is that the former is more useful to measure traffic on links. If we say that each node sources one unit of messages to the network, it sources ψ_k units to each destination a distance k away. The sum of ψ_k over all destinations is unity, and multiplying ψ_k by the total output of a source gives the traffic along each path $(s\ d)$, where d is distance k from s. It is important to note that ψ is determined assuming a symmetric, that is, fault-free, hypercube. This is a valuable assumption, since application programs on the system are written under the same assumption. In our analysis we must map this ψ onto the degraded hypercube links. We use the term distance of a path $(s\ d)$ to describe the number of bits different between s and d (the Hamming distance). The length of a path is the number of links which are actually traversed by a message from s to d in configuration F. The length is always greater than or equal to the Hamming distance.

3.2. Traffic Analysis

We now use the above quantification of destination distribution in an analysis of traffic on each link in a possibly degraded hypercube. The traffic, or throughput, on a link is a measure of how many messages cross the link in one time unit. Throughout this thesis, we use as our basic time unit the mean time of actual message transmission, not including path setup time.

For each link l we define a vector $v(l)$ with n components, where n is the number of dimensions as well as the maximum distance in the hypercube. Component k of the vector $v(l)$ corresponds to the number of paths on l which are of distance k. We will describe later how v can be derived. We will give an analytical paradigm in the case of single failures, and we will give an algorithm to find ν in the case of more complicated configurations of faulty hypercubes.

The relative traffic on l, $t(l)$, is equal to the dot product of $\nu(l)$ and ψ in units of total messages output by one node. The arrival rate of messages, or absolute traffic, on l, $\lambda(l)$, is found by multiplying the traffic $t(l)$ by a factor λ^*, the messages per time unit sourced by one processor.

It is interesting to note that traffic here is composed primarily of two components, ψ and $\nu(l)$. ψ, describing the selection of message destination, is a function of the application program. $\nu(l)$, on the other hand, is exclusively a function of hypercube configuration. The product of these two vectors gives us traffic.

In Kleinrock [31] an analytical method is given which finds traffic $t(l)$ on each link l from the routing matrix; element (l, l') of the routing matrix is the probability of going from link l to link l'. Path Analysis differs from this approach in that, due to the possibly degraded condition of the network, we do not know the routing matrix. We find the routing matrix from the traffic, after the traffic is derived from paths.

First we define $N(l)$ by expanding the vector $\nu(l)$. $N(l)$ is an M by n matrix, where M is the number of links in the hypercube. Element (l', k) in $N(l)$ is the number of paths of distance k which cross link l' immediately after crossing link l. Multiplying $N(l)$ by ψ gives the M-vector $s(l)$. The scalar measurement of traffic $t(l)$ is closely related to $s(l)$. $t(l)$ is the sum of the elements of $s(l)$ plus the traffic from those paths for which l is the last link traversed. Thus $r(l)=s(l)/t(l)$ is the routing vector for link l:

component l' of $r(l)$ gives the probability that an arbitrary message on l has l' as its next link. The M by M matrix **R** is formed row by row from $r(l)$.

R is the routing matrix for the hypercube. Note that **R** has zeros along the diagonal since it is impossible for a message to have as its next link its current link. The sum of the elements in row l of **R** ($r(l)$) is a number whose difference from unity is the probability that l is the terminating link of a message's path.

3.2.1. Example

As an example of the above calculations, we will derive $\nu(l)$, $t(l)$, $s(l)$, and **R** for a fault-free four-dimensional hypercube. For ease of calculation we assume ψ to be uniform. Also, since all links of a given dimension are identical in a perfect cube, we let l represent not a link address, but rather a dimension. We characterize each links by its dimension and do our analysis relative to this characterization.

Along with ψ, we give $\nu(l)$ for a perfect cube as derived later in this thesis (there called $\alpha(l)$).

$$\nu(l) = \begin{bmatrix} 1 \\ 3 \\ 3 \\ 1 \end{bmatrix} \qquad \psi = \begin{bmatrix} 1/15 \\ 1/15 \\ 1/15 \\ 1/15 \end{bmatrix}$$

Multiplying these two yields $t(l) = \nu(l) \cdot \psi = 8/15$. This is the total traffic on each link of a perfect four-dimensional hypercube under *ecube* routing.

Now we give the simplified matrices $\mathbf{N}(l)$. They are simplified because we are characterizing l by its dimension rather than its address. Thus, where $\mathbf{N}(l)$ is generally a 64 by 4 matrix, we make it a 4 by 4. Remember that element (l', k) of $\mathbf{N}(l)$ is the number of paths of length k crossing link l' immediately after crossing link l. (In this example, it is the number of paths of length k crossing a link of dimension l' immediately after crossing a link of dimension l.)

$$N(0) = \begin{bmatrix} 0&0&0&0\\ 0&1&2&1\\ 0&1&1&0\\ 0&1&0&0 \end{bmatrix} \quad N(1) = \begin{bmatrix} 0&0&0&0\\ 0&0&0&0\\ 0&1&2&1\\ 0&1&1&0 \end{bmatrix} \quad N(2) = \begin{bmatrix} 0&0&0&0\\ 0&0&0&0\\ 0&0&0&0\\ 0&1&2&1 \end{bmatrix} \quad N(3) = \begin{bmatrix} 0&0&0&0\\ 0&0&0&0\\ 0&0&0&0\\ 0&0&0&0 \end{bmatrix}$$

N(3) = 0 because no paths on a third dimension link continue to another link.

We multiply N(l) by ψ to get $s(l)$.

$$s(0) = \begin{bmatrix} 0\\ 4/15\\ 2/15\\ 1/15 \end{bmatrix} \quad s(1) = \begin{bmatrix} 0\\ 0\\ 4/15\\ 2/15 \end{bmatrix} \quad s(2) = \begin{bmatrix} 0\\ 0\\ 0\\ 4/15 \end{bmatrix} \quad s(3) = \begin{bmatrix} 0\\ 0\\ 0\\ 0 \end{bmatrix}$$

Dividing each $s(l)$ by $t(l)$ (from above, 8/15 for all l) to get $r(l)$, we form the matrix **R** with row 1 equal to $r(0)^T$, row 2 equal to $r(1)^T$, and so on.

$$\mathbf{R} = \begin{bmatrix} 0 & 1/2 & 1/4 & 1/8\\ 0 & 0 & 1/2 & 1/4\\ 0 & 0 & 0 & 1/2\\ 0 & 0 & 0 & 0 \end{bmatrix}$$

We can verify with a simple probability analysis that this is indeed the routing matrix for the perfect four-dimensional hypercube. If we view each bit as having a 1/2 chance of being changed in a particular route, element (l, l') in **R** is the probability that l' is the next highest bit after l which is changed in the route. The result is that the number $l-l'$ is chosen from a binomial distribution.

The above example is greatly simplified compared to the general case. But the traffic and routing constructs we have developed apply to any arbitrary faulty condition of the hypercube. We now use these traffic and routing matrices to perform a delay analysis for the hypercube in its arbitrarily faulty state.

3.3. Delay Analysis

To begin our section on delay, we briefly discuss the analysis for delay in a packet-switched network as presented in [31]. This analysis is made under the assumptions of exponential message arrival with mean frequency $\lambda(l)$ and exponential message service time with mean $\bar{y}$ identical on all links. Given the $\lambda(l)$ as we calculated in the preceding section, the delay on link l is, from [31],

$$w(l) = \frac{1}{\frac{1}{\bar{y}} - \lambda(l)}$$

Since here we assume packet-switching, the total delay for the transmission of a message is the sum of the delays of all the links the message traverses. That is, on a path $(s\ d)$ the total delay for a message is

$$T(s\ d) = \sum_{l \text{ in } (s\ d)} w(l)$$

Now we will derive the expressions for delay assuming a circuit-switched network. As mentioned before, delay analysis of circuit-switched networks is not popular in the literature. The difficulty in analysis of circuit-switching over packet-switching is in quantifying the holding of a link while waiting for a later link to free up. Starting with the most general terms, we have three random variables for link l: y is the transmission time, the time the message is actually being sent; x is the service time, the time the message is being sent plus the time it must hold to await the freeing of a later link; w is the response time, the delay, the sending time plus the holding time plus the time waiting for

l itself. Let $f_x(u)$ be the probability density function (pdf) of x, $f_y(u)$ be the pdf of y, and $f_w(u)$ be the pdf of w.

We use the mixture distribution [32] to reconcile these three varieties of random variables over the many links. Mixture distributions are of the form

$$F(u) = \sum_{i=1}^{i=m} r_i F_i(u),$$

where the sum of the m numbers r_i is 1. Our service time x is one of a number of values, each with a certain probability. If link l is the terminating link for the message, then the service time on l for the message is simply $x(l)=y$. If, on the other hand, the message proceeds from l to l', the service time on l is equal to the delay on l', $x(l)=w(l')$. That is, the time l is servicing the message is equal to the time the message must wait for service from l' plus the actual service on l'. We have already developed $\mathbf{R}(l,l')$ as the routing matrix for the network. We use these probabilities to determine $f_{x_l}(u)$:

$$f_{x_l}(u) = (1 - \sum_{l'} \mathbf{R}(l,l'))f_y(u) + \sum_{l'} \mathbf{R}(l,l')f_{w_{l'}}(u)$$

Let x and w be the obvious M-vectors. Let $\bar{y}$ be the mean message transmission time. Let y be the M-vector with all components equal to $\bar{y}$, with the inherent assumption that all links and messages are identical in this regard. In the language of the matrices we have created,

$$x = (\mathbf{I}-\mathbf{R})y + \mathbf{R}w = y + \mathbf{R}(w-y) \qquad (1)$$

This expression for x is a function of y and w. y is a random vector of arbitrary distribution. w is a random vector dependent on x.

We must make some simplifying assumptions to come up with a reasonable expression for w. One of these assumptions is that message arrival on a link is exponential. The other assumption is that message service time is exponential. When we assume these two facts, we can examine average service times only, without requiring discovery of service time distributions. For an M/M/1 queue, the average delay $\bar{w}$ is related to the average service time $\bar{x}$ and the average arrival frequency λ through the expression

$$\bar{w} = \frac{1}{\frac{1}{\bar{x}} - \lambda}$$

For our approximation of delay, w, we have the M-function

$$w(l) = \frac{1}{\frac{1}{x}(l) - \lambda(l)}$$

This w we put into Equation (1), and we iterate, from $x^0 = 0$, to find the fixed point for x. Calculating w from this x gives us the delay, the vector we are seeking.

The delay for a message is found very easily from w. Since a link is held for the duration of the transmission of a message across it, the delay of a message is equal to the delay of the first link it uses. If l is the first link in a path $(s\ d)$, then the average delay for a message taking $(s\ d)$ is $w(l)$.

This analysis needs only one set of inputs to drive it. Only one set of parameters comes from the degraded hypercube and reroute methods. These parameters are the vectors $\mathrm{v}(l)$ and the matrices $\mathbf{N}(l)$, for each l. We need to have a way to derive v and $\mathbf{N}$. In the following section we introduce an intuitive analytical method to obtain the numbers of paths on each link. Then we present an easily usable computational method to find v and $\mathbf{N}$.

3.4. Path Analysis

We continue the theoretical section with a discussion of the analytic determination of the numbers of paths of length k, the vector ν. In the next section we give an algorithm to determine these numbers quickly for use in the formula derived above.

In deterministic routing procedures, certain rules apply to the paths which cross a link l. By recognizing and following these rules, we can determine exactly what paths go on each link, and thus determine $\nu(l)$. Recall that $\nu(l)_k$ is the number of paths of length k which cross l. We can break ν into three terms: α, β, and γ. $\alpha(l)$ is the vector of paths which are on l in a fault-free cube. $\beta(l)$ is the number of paths which are removed from l due to a fault. $\gamma(l)$ is the number of paths which are put on l due to rerouting the failed paths. Thus we have

$$\nu(l) = \alpha(l) - \beta(l) + \gamma(l)$$

3.4.1. Calculation of $\alpha(l)$

In order to determine $\alpha(l)$, the vector of paths which are on l due to *ecube* in a fault-free hypercube, we must thoroughly analyze the standard *ecube* routing. *Ecube* routing dictates that the bit differences from source to sink are resolved from right to left. If l is along the m^{th} dimension, then all bits to the right of m have been resolved, and all bits to the left of m have not been resolved. The m^{th} bit of the address is currently being resolved on l. In useful notation, let us look at the bits of l. Let ι signify an arbitrary bit of l (context tells us which bit). The address of l is the node the link exits from; the dimension that l traverses is identified in our notation by placing a hat over that bit. Let

υ be an unrestricted bit in the source or destination, a bit which can be either 0 or 1. We call υ a free or unbound bit.

$$\begin{aligned} l &= \iota \,.\,.\, \iota\, \hat{\iota}\, \iota \,.\,.\, \iota \\ s &= \iota \,.\,.\, \iota\, \underline{\iota}\, \upsilon \,.\,.\, \upsilon \\ d &= \upsilon \,.\,.\, \upsilon\, \bar{\iota}\, \iota \,.\,.\, \iota \end{aligned}$$

Since we have $n-1-m$ bits to the left of m which are unknown in the destination and m bits to the right of m which are unknown in the source, there are a total of $n-1$ bits which can vary in the path $(s\ d)$. 2^m different sources and 2^{n-1-m} different destinations make 2^{n-1} different paths which cross link l in a perfect *ecube*-routed hypercube, regardless of which l we choose.

Similarly we can describe exactly the number of paths of length k which cross l. There are $n-1$ bits in $(s\ d)$ which determine the distance between s and d. We already know there is one bit different between s and d, the m^{th} dimension, which link l traverses. Each other bit which is different is one more dimension of difference between s and d. Therefore, the number of paths of length k crossing l is the number of ways we can choose $k-1$ bits from $n-1$ bits; we call this number in the case of *ecube* routing in a perfect hypercube $\alpha(l)_k$ where

$$\alpha(l)_k = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$$

Here we have clear demonstration of the network symmetry that *ecube* provides. Every link has exactly the same cross section of traffic by path distance as every other link.

3.4.2. Calculation of β(*l*)

The vector β arises from the fact that, in a faulty cube, a certain set of paths is unroutable using *ecube*. If a path from this set crosses *l*, then, without a reroute strategy, *l* will have one less path using its facilities. We can illustrate the calculation of β(*l*) with an example fault. Due to the symmetry of the cube, any single fault we cite will be said to be addressed at node 0.

As an example of determining β(*l*), let us assume a single link failure. Let the fault be in the m^{th} link from node 0 (both entry and exit links). We have three possible cases to examine the paths lost from link *l*: *l* is traversed with *ecube* by some paths before the faulty link, *l* is traversed by some paths after the faulty link, or *l* is the faulty link. The key point of the above three cases is that we are trying to find all paths which cross *l* and cross the faulty link under *ecube* routing. If *l* does not fall into one of these categories, it does not lose any paths due to the faulty link, and β(*l*) = 0. Our notation below is similar to that above, but we indicate the dimension of the faulty link with an arrow.

Case 1:

$$\begin{array}{l}
l = 0\,.\,.\,0\,\iota\,\iota\,.\,.\,\iota\,\hat{1}\,0\,.\,.\,0 \\
s = 0\,.\,.\,0\,\underline{\iota}\,\iota\,.\,.\,\iota\,1\,\upsilon\,.\,.\,\upsilon \\
d = \upsilon\,.\,.\,\upsilon\,\bar{\iota}\,0\,.\,.\,0\,0\,0\,.\,.\,0 \\
\uparrow
\end{array}$$

Case 2:

$$\begin{array}{l}
l = 0\,.\,.\,0\,\hat{0}\,\iota\,.\,.\,\iota\,\iota\,0\,.\,.\,0 \\
s = 0\,.\,.\,0\,0\,0\,.\,.\,0\,\underline{\iota}\,\upsilon\,.\,.\,\upsilon \\
d = \upsilon\,.\,.\,\upsilon\,1\,\iota\,.\,.\,\iota\,\bar{\iota}\,0\,.\,.\,0 \\
\uparrow
\end{array}$$

Case 3:

$$\begin{array}{l}
l = 0\,.\,.\,0\,\hat{\iota}\,0\,.\,.\,0 \\
s = 0\,.\,.\,0\,\underline{\iota}\,\upsilon\,.\,.\,\upsilon \\
d = \upsilon\,.\,.\,\upsilon\,\bar{\iota}\,0\,.\,.\,0 \\
\uparrow
\end{array}$$

An important feature in the above cases is what can be called a *run* of the fault address. The fault here is 0..0-0..0. It appears in all these cases as follows: we start at the left of s; we read over to the m^{th} bit and find all 0's; the m^{th} bit in the source is l and in the destination is $\bar{l}$; the rest of the bits in the destination are 0. We see that a run is formed where we can read from the path $(s\ d)$ the faulty link.

We can do the same sort of bitwise analysis for the cases of a single fault in processor 0 or a fault which destroys all of node 0. To determine $\beta(l)$ from this derivation of paths crossing l and the fault, we again examine the Hamming distance between the $(s\ d)$ pairs we have found. In the above example the three cases are mutually exclusive. Given the l we are characterizing with $\beta(l)$, we need to decide which case it falls under, an easy task here. For Case 3, $\beta(l)=\alpha(l)$, since every path crossing the faulty link is lost. We will concentrate on Cases 1 and 2; let $H(s\ d)$ be the Hamming distance between s and d for a certain assignment of the unknown bits in $(s\ d)$. In Cases 1 and 2, $H(s\ d)\geq 2$ because two links must be crossed (one is l, and one is the faulty link). For a specific l, $H(s\ d)\geq r$, where r is 2 plus the number of bits between the dimension of l and the dimension of the faulty link which differ between s and d. This r represents the minimum k for which $\beta(l)_k$ is not zero. If there are q bits free (q υ's) in $(s\ d)$, then

$$\beta(l)_k = \binom{q}{k-r}$$

3.4.3. Calculation of $\gamma(l)$

The value of α depends only on the *ecube* route. The value of β depends only on the *ecube* route and the specific fault. The value of γ, the number of paths placed on a link because of the reroute technique, is the only number depending on reroute strategy.

Breaking down the cases for γ is difficult, but not impossible for a single link or node fault. Note that for a processor failure, where communication is still possible through the node, no reroute takes place and γ is zero. We will demonstrate the analysis of $\gamma(l)$ using a node fault, showing some of the differences between node faults and link faults.

Assume node 0 is faulty, that is, no paths may be routed through node 0. To figure $\gamma(l)$ we need to determine those paths $(s\ d)$ that *ecube* routes through node 0 but our reroute strategy routes across link l. In this example, let the reroute strategy be *C&S adaptive* [27] discussed earlier in Section 2.2. First, a number of things need to be noted about *C&S adaptive*. Under a single node failure, no spare dimensions are taken by *C&S adaptive*. A spare is taken only if all the dimensions yet to be resolved are faulty. Therefore, a single fault hypercube will only force a path to take a spare dimension if the last link to be covered is faulty. This is impossible since any path with a faulty last link would have as its destination processor 0, which is faulty itself. In effect the *C&S adaptive* reroute method, with a single node failure at node 0, makes one modification to the normal *ecube* routing method: it swaps the order a path resolves the dimensions by which it would normally enter and exit node 0, thus bypassing node 0.

The $(s\ d)$ pairs we have to find can be broken up into the following functional cases:

Case 1:	path not yet reached node 0
Case 2a:	path hit 0, now resolving next highest dimension
Case 2b:	path passing 0 by dimension of conflict
Case 2c:	path back to normal *ecube* in higher dimensions

We now discuss each case separately.

Case 1:

$l = 0\,.\,.\,01\iota\,.\,.\,\iota\hat{1}0\,.\,.\,0$
$s = 0\,.\,.\,01\iota\,.\,.\,\iota1\upsilon\,.\,.\,\upsilon$
$d = \upsilon\,.\,.\,\upsilon00\,.\,.\,000\,.\,.\,0$

In Case 1 we notice the run of the faulty component. The first rule in (s d) derivation is that routing by *ecube* must take the path (s d) through the fault. Otherwise, the path does not require rerouting.

Case 2a:

$l = 0\,.\,.\,0\hat{0}0\,.\,.\,010\,.\,.\,0$
$s = 0\,.\,.\,000\,.\,.\,01\upsilon\,.\,.\,\upsilon$
$d = \upsilon\,.\,.\,\upsilon10\,.\,.\,000\,.\,.\,0$

This case is for the value of l which we take in order to avoid node 0. Resolving the 1 which appears in the link address would have made the path through node 0. We instead must resolve the next highest bit between s and d. (Note the space of 0 bits in both s and d.)

Case 2b:

$l = 0\,.\,.\,010\,.\,.\,0\hat{1}0\,.\,.\,0$
$s = 0\,.\,.\,000\,.\,.\,01\upsilon\,.\,.\,\upsilon$
$d = \upsilon\,.\,.\,\upsilon10\,.\,.\,000\,.\,.\,0$

All of these cases still have the run of 0's which must exist for this path to normally have gone through node 0. In this case we have resolved the next highest bit, and now are resolving the bit which would have otherwise carried us into node 0.

Case 2c:

$l = 0\,.\,.\,0\hat{0}\iota\,.\,.\,\iota00\,.\,.\,0$
$s = 0\,.\,.\,000\,.\,.\,00\upsilon\,.\,.\,\upsilon$
$d = \upsilon\,.\,.\,\upsilon1\iota\,.\,.\,\iota10\,.\,.\,0$

Case 2c is the same as Case 1, but for a link on the other side of the faulty node.

Calculating $\gamma(l)$ from these cases is more difficult than calculating $\beta(l)$. These functional cases we have derived are exclusive, but they are not independent. In fact, any link which works in Case 2b also works in Case 1, and any link which satisfies Case 2c satisfies 2a as well. In these instances we simply add the numbers derived from each of the two cases because no specific path which appears in one case appears in the other.

Let us use Case 1 as an example in computing $\gamma(l)_k$. Again, there are r bits forced to be different between s and d because of the case and the address of link l. We need to choose $k-r$ from the q unbound bits to find $\gamma(l)_k$. But there is another factor to consider here. There is one assignment of the q_d unbound bits in d which causes d to equal zero. This is impossible since node 0 is faulty and can source or sink no communication. We must subtract off the number of ways that all $k-r$ bits can be chosen from the q_s free bits of s: at least one of these $k-r$ bits must appear in d. Thus, for an l which satisfies Case 1 (and not Case 2b),

$$\gamma(l) = \begin{bmatrix} q_s+q_d \\ k-r \end{bmatrix} - \begin{bmatrix} q_s \\ k-r \end{bmatrix}$$

Getting numbers for the other cases is similar, always avoiding a faulty source or destination.

One thing which should be noted is that in all these calculations, the sum over all k of α, β, or γ equals what we compute when we count total paths using the numbers of unbound bits. In the above example for $\gamma(l)$, the total number of paths $(s\ d)$ which are rerouted onto l is

$$2^{q_s+q_d}-2^{q_s}=\sum_k\left[\binom{q_s+q_d}{k-r}-\binom{q_s}{k-r}\right]$$

3.4.4. Example application of path analysis

As an example using all of α, β, and γ to calculate ν, we look at the following case in a four-dimensional cube: node 0000 is faulty, the reroute is *C&S adaptive*, and the link we are examining is $l = 001\hat{1}$. Here for the sake of simplicity, we compress $\nu(l)$ into a scalar quantity, ignoring k and counting all paths the same regardless of their length.

First we look at $\alpha(l)$, the number of paths on l in a perfect hypercube.

$$\begin{array}{lcl}
l & = & \iota\,.\,.\,\iota\,\hat{\iota}\,\iota\,.\,.\,\iota & = & 001\hat{1} \\
s & = & \iota\,.\,.\,\iota\,\underline{\iota}\,\upsilon\,.\,.\,\upsilon & = & 0011 \\
d & = & \upsilon\,.\,.\,\upsilon\,\iota\,\iota\,.\,.\,\iota & = & \upsilon\upsilon\upsilon 0
\end{array}$$

From this layout, we see that there are three unknown bits in $(s\ d)$. Thus our scalar $\alpha(l) = 2^3 = 8$.

We need to create the proper layout to find $\beta(l)$. We are trying to find those paths which cross l and go through node 0000. By writing l, s, and d as in the other layouts, we can find the paths $(s\ d)$ which apply.

$$\begin{array}{lcl}
l & = & 001\hat{1} \\
s & = & 0011 \\
d & = & \upsilon\upsilon 00
\end{array}$$

Note the run of 0000 in s and d. Bit 0 goes from 1 to 0 in $(s\ d)$ because that is precisely the action performed by l. Bit 1 goes from 1 to 0 because otherwise the *ecube* paths described would not be routed through faulty node 0. Bits 2 and 3 are resolved after both l and node 0000 and are therefore unbound in the destination. The scalar $\beta(l)$ calculated from the above determination of the required $(s\ d)$ is $\beta(l) = 2^2 = 4$.

The link and fault combination in our example matches those for Cases 1 and 2b of the $\gamma(l)$ discussion above. We thus copy the layouts from above and substitute for l.

$$\begin{array}{lcl} l = 0\,.\,.\,0 1 \iota\,.\,.\,\iota \hat{1} 0\,.\,.\,0 & = & 0 0 1 \hat{1} \\ s = 0\,.\,.\,0 1 \iota\,.\,.\,\iota 1 \upsilon\,.\,.\,\upsilon & = & 0 0 1 1 \\ d = \upsilon\,.\,.\,\upsilon 0 0\,.\,.\,0 0 0\,.\,.\,0 & = & \upsilon \upsilon 0 0 \end{array}$$

$$\begin{array}{lcl} l = 0\,.\,.\,0 1 0\,.\,.\,0 \hat{1} 0\,.\,.\,0 & = & 0 0 1 \hat{1} \\ s = 0\,.\,.\,0 0 0\,.\,.\,0 1 \upsilon\,.\,.\,\upsilon & = & 0 0 0 1 \\ d = \upsilon\,.\,.\,\upsilon 1 0\,.\,.\,0 0 0\,.\,.\,0 & = & \upsilon \upsilon 1 0 \end{array}$$

The contribution to $\gamma(l)$ of the $(s\ d)$ from Case 1 is 3. One apparent $(s\ d)$ path has d=0, and it is not counted in $\gamma(l)$. The other three paths in Case 1 were counted as lost in calculating $\beta(l)$. *C&S adaptive*, however, would route those three across link l anyway, taking care of rerouting them later. There are 4 paths to be added to $\gamma(l)$ from Case 2b. These paths are those which start at node 0001 and want to go to or through 0010. Due to the fault, they cannot proceed through 0000, and thus are rerouted through 0011 to 0010.

The total number of paths $\nu(l)$ on link $l = 001\hat{1}$ due to a faulty node 0000 and *C&S adaptive* rerouting is

$$\nu(l) = \alpha(l) - \beta(l) + \gamma(l) = 8-4+(3+4)=11$$

This example considered ν to be scalar for ease of presentation, but the extension to the vector ν is clear and straightforward.

3.5. Program Implementation of Path Analysis

The above mathematical determination of ν is only possible for a single fault in the entire structure of the hypercube. More than one fault excessively complicates the cases and the counting. However, we have developed and applied an algorithm which very simply and quickly determines ν and $\mathbf{N}$ for any faulty condition of the hypercube network.

The algorithm for finding the paths crossing the links is given below.

```
ALGORITHM FIND-PATHS
   Create a graph of the degraded hypercube, identifying faulty components
   For every node $s$
      For every other node $d$
         Try routing ($s$  $d$) by *ecube*
         If *ecube* fails, try reroute strategy
         If *ecube* and reroute strategy fail, the path fails
      Endfor
   Endfor
```

For each successful path, we check the links it crosses and take whatever counts we need. Namely, we store up the routing count matrix $\mathbf{N}(l)$ and the vector $\nu(l)$ for every l in the hypercube. From these counts, we can determine the arrival rate λ and the delay w for each link l as shown previously.

3.6. Reconfiguration Modeling

As mentioned before, our analysis models the remapping of tasks as proposed by Banerjee [26]. Specifically, we view a task on a node as being divisible into n similar processes. After a processor fails, its task is divided as similar processes among the n neighboring nodes of that processor. These *displaced processes* still behave, communication-wise, as though they were united on the now-faulty processor.

To account for the redistribution of tasks in this fashion, only minor changes need to be made to the analysis we have thus far presented. This is because the communication due to the redistributed work is merely superimposed upon the traffic already existing due to unaffected processors. Therefore, the traffic and delay analyses are unaffected. We have previously handled the loss of paths due to a processor fault by subtracting the appropriate contributions from ν and N; for ν the term for lost paths was β. Similarly, we can model task redistribution with appropriate additions to ν and N.

Consider ν^R and N^R. These are analogous to ν and N, but consist exclusively of traffic due to task redistribution. As discussed above, we look at all the paths $(s\ d)$ which cross a link l. If the s or d determined to be an endpoint of a path is adjacent to a faulty processor, that path carries traffic from a displaced process. We account for this appropriately in ν^R and N^R.

We will describe the formulation of $\nu^R(l)$; the extensions of this to $N^R(l)$ are straightforward, essentially recognizing and recording the order of links in each path. Recall that $\nu(l)_k$ holds the number of paths of distance k crossing link l. $\nu^R(l)_k$ carries the number of *virtual* paths of length k crossing link l due exclusively to task reconfiguration. Since each displaced process is $1/n$ of the standard processor task, we assume it sources and sinks $1/n$ times the number of messages a standard processor would. If we find that $(s\ d)$ carries traffic for a displaced process across l, we add $1/n$ to the appropriate element of $\nu^R(l)$. The appropriate element k is not determined from the distance between s and d, as it is in $\nu(l)$. The correct k in $\nu^R(l)$ is the distance between the original source and destination, that is, k is calculated as though the displaced process is not displaced. In this way we preserve the original destination distribution intended by the application program while still considering the remapping of tasks from faulty computational units.

As we have presented the task redistribution theory, we ignore certain second-order effects. Among these is the failure of two adjacent processors. Also, we do not account for the traffic due to communication which may be required among processes displaced from the same node. The latter of these could easily be modeled by including a ψ_0 which describes the amount of communication among processes on one processor. When these processes are displaced, they still must communicate, but now across links instead of through memory. These effects we can include in $\nu^R(l)$ and $N^R(l)$, although we have not included these in our implementation because of the specific nature of ψ_0.

By adding $\nu^R(l)$ to $\nu(l)$ and $N^R(l)$ to $N(l)$, we can proceed with the analyses of traffic and delay as presented previously in this thesis.

CHAPTER 4.

COMPARISON OF PATH ANALYSIS WITH SIMULATION

We have written a program which implements Path Analysis using algorithm FIND_PATHS described above. We compared the results given by the Path Analysis program with those given by simulation. For simulation we used the CSIM simulation environment [33]. In the simulation we chose each source to have an identical exponential intermessage frequency of mean λ^* and each link to have an identical exponential service time mean 1. Thus the times we observe are in units of average message transmission time. Traffic is in units of number of messages per mean transmission time, and delay is in units of mean transmission time.

We show results for a four-dimensional fault-free hypercube (F_0) and a four-dimensional hypercube (F_4) with four faults: node 0000, node 0011, link 1-01, and link -110. For each of these, we display graphs for a uniform destination distribution and a sphere of locality (r=1 and ϕ=.9) distribution. Figure 2 shows F_0 under uniform distribution, Figure 3 shows F_0 under spherical distribution, Figure 4 shows F_4 under uniform distribution, and Figure 5 shows F_4 under spherical distribution. Each figure is a quartet of graphs displaying theoretical and simulation results for traffic and delay for the various links. The 64 links are grouped by dimension with the lower dimensions to the left.

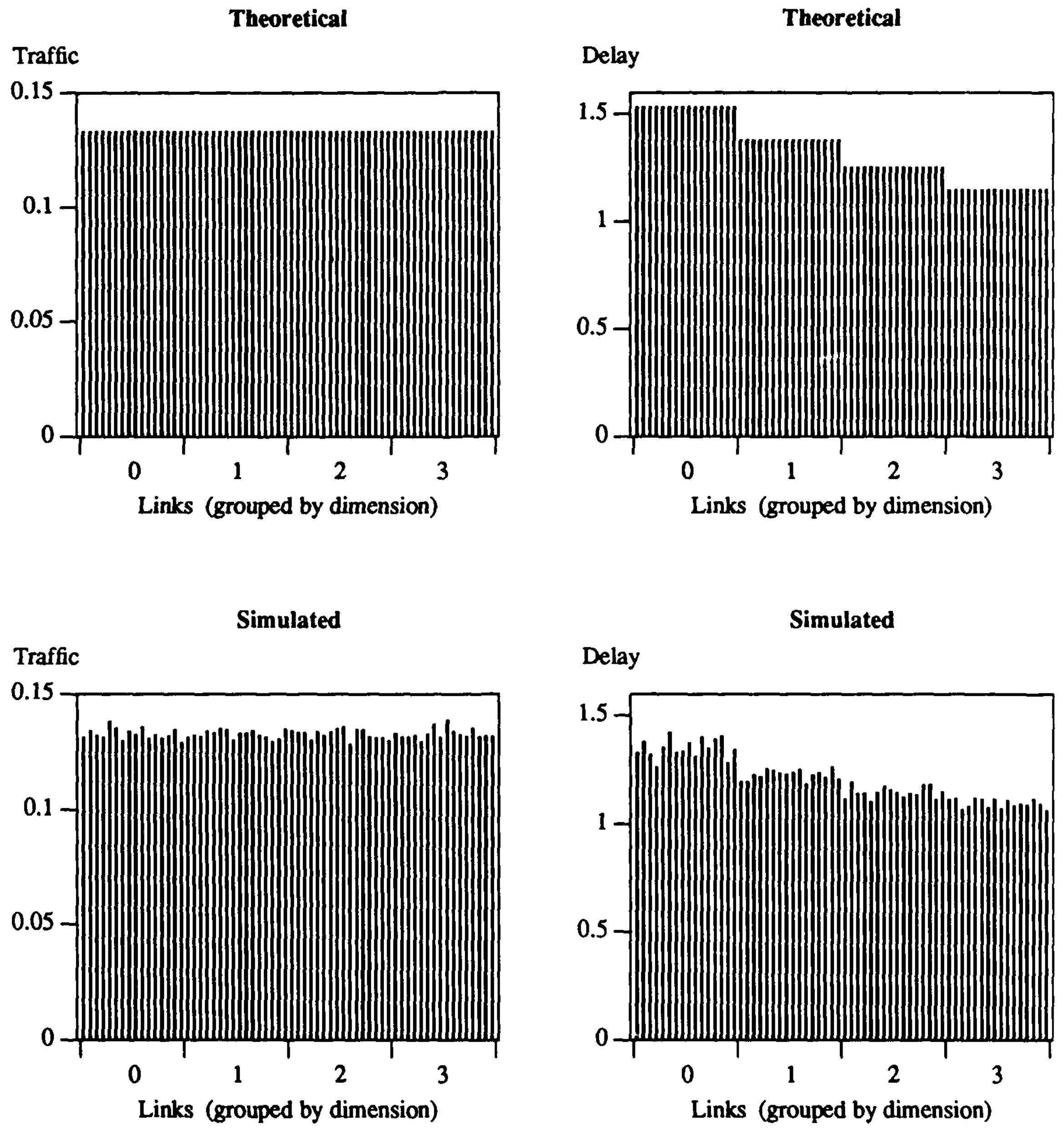


Figure 2. Path Analysis Compared with Simulation:
Fault-Free Hypercube, Uniform Destination Distribution

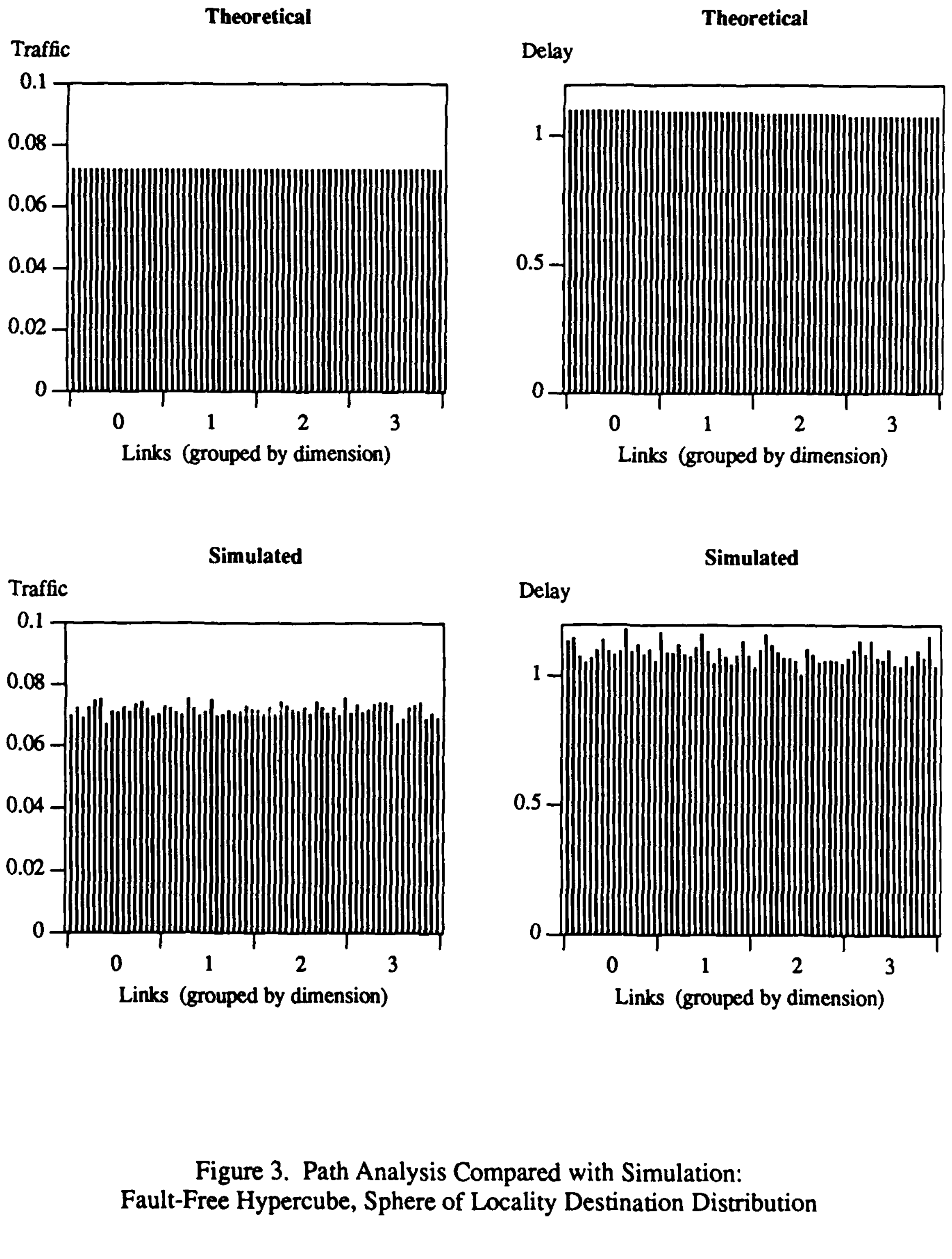


Figure 3. Path Analysis Compared with Simulation:
Fault-Free Hypercube, Sphere of Locality Destination Distribution

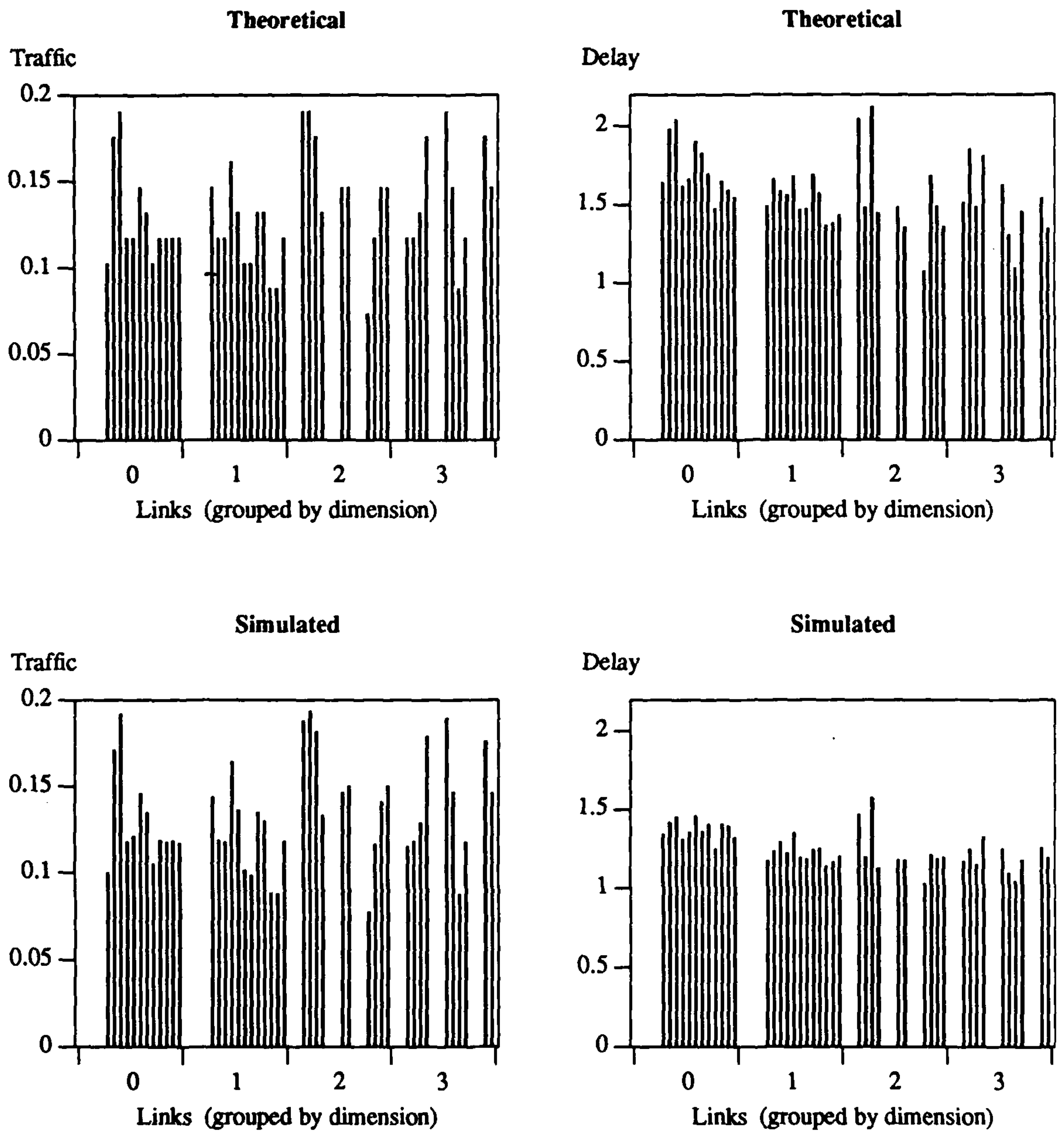
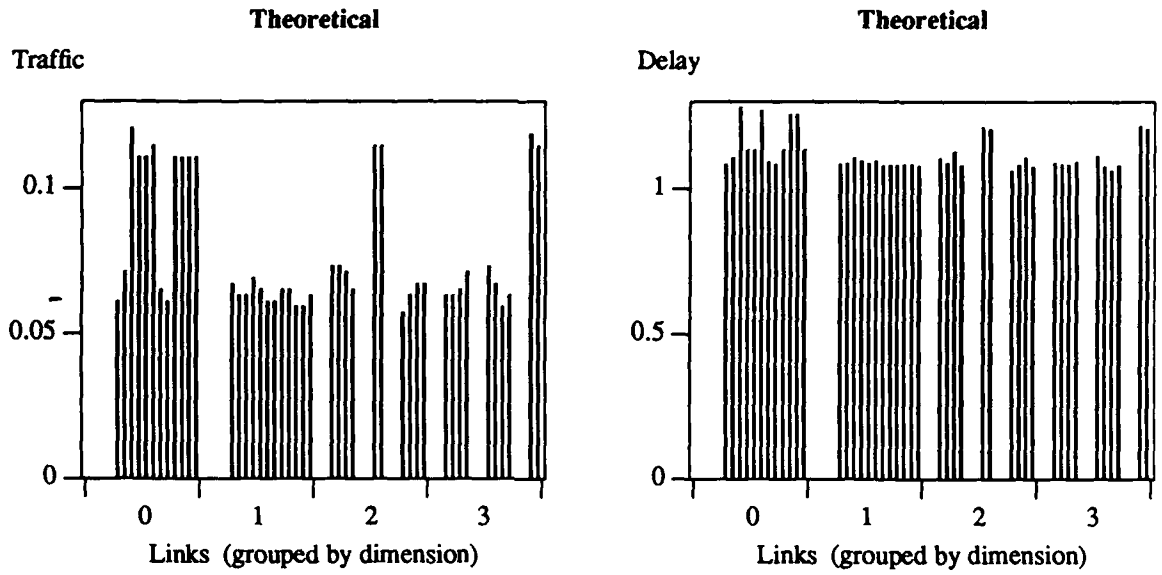


Figure 4. Path Analysis Compared with Simulation:
Four-Fault Hypercube, Uniform Destination Distribution



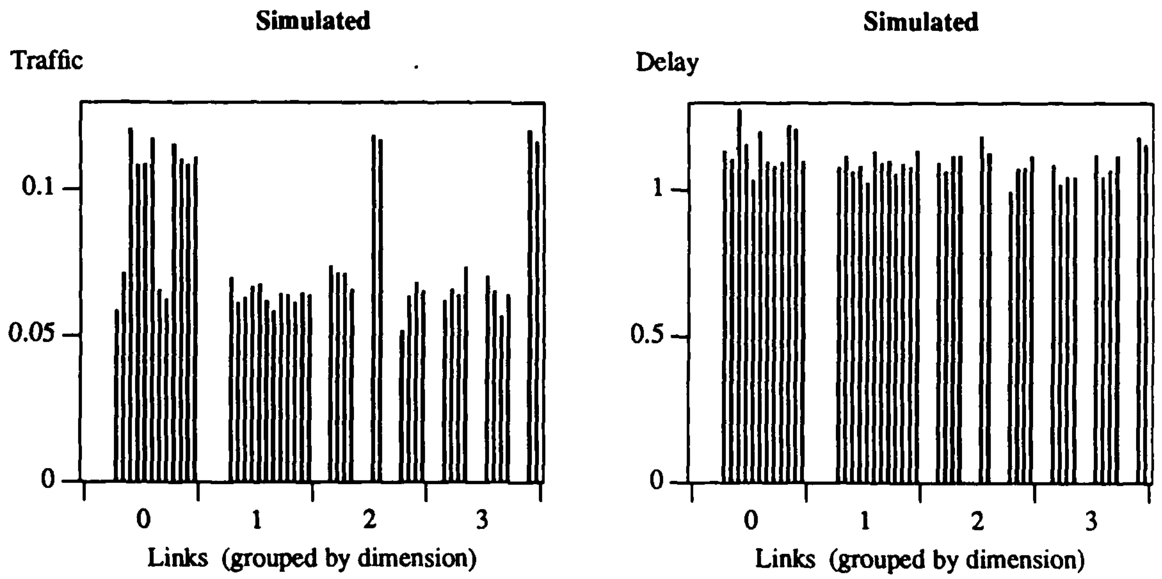


Figure 5. Path Analysis Compared with Simulation:
Four-Fault Hypercube, Sphere of Locality Destination Distribution

The graphs show us that the theoretical estimate of the traffic, or throughput as CSIM calls it, is essentially exact for all cases. We expect this similarity because there are no approximations made in our analysis of traffic. The only deviation from reality here is the probabilistic characterization of destination.

The accuracy of delay, however, varies dramatically. The first fact we should note is that our model gives much better delay results for a sphere of locality distribution than for a uniform distribution. The effects which cause the simulated system to deviate from our delay model are lengthy paths. The more single-hop messages in the system, the more exponential the arrivals and the waits for link service become. Since a dominance of single-hop paths is common in hypercube applications, we consider the model's accuracy in predicting these delays to be very valuable.

We can see from the graph of the perfect cube with uniform destination distribution in Figure 2 that the approximation for delay grows worse from higher dimensions to lower dimensions. The weakest part of the theoretical delay calculations is that of the delay from the utilization. Since a higher dimension link in a perfect hypercube does not have to incorporate into its own utilization the delay experienced in later links, it has a more accurate theoretical delay than a lower dimension link. That the delay is as erroneous as it is for the highest dimension links is a testament to the inaccuracy of the assumptions. By the time messages have made their way from the lower links to the highest, their arrival is hardly exponential. These effects, as well as assuming the general service distributions to be exponential, cause the theoretical model to consistently overestimate the simulated results for delay.

We have provided Table 1 for quantitatively comparing theoretical and simulated delay results for the four cases. We give average delay over all the usable links for both theoretical and simulated runs, as well as the percentage deviation of theoretical from simulated results. We see that the model is very close to the simulated results for the sphere of locality destination distribution. It is also evident that the delay model has some difficulty dealing with uniform ψ in the highly faulty network, although the approximation for sphere of locality is still very good.

The value of Path Analysis is clear when we see that it gives the correct traffic estimation, regardless of assumed intermessage frequency distribution or service time distribution. Although the delay is overestimated in our model, slightly or greatly depending on destination distribution, it is consistently overestimated and correctly reflects the profile of the actual delay. We can use the program's exact traffic calculations to recognize which links saturate first under different degraded configurations, and we can see how badly assorted faults increase the message delay for faulty hypercubes with reroute strategies.

Table 1. Comparison of Theoretical and Simulation Results.

Average Link Delay				
Dest. Dist.	Configuration	Theory	Simulation	%difference
uniform	no faults	1.3324	1.2073	+10.36
uniform	4 faults	1.5834	1.2609	+25.58
sphere	no faults	1.0912	1.0919	-0.06
sphere	4 faults	1.1204	1.1065	+1.26

Of course, the most important feature of the Path Analysis program is that *it is fast*, much faster than the simulations we ran. The actual difference in execution time was quite variable in our tests, but the runs of the Path Analysis program were generally well over 100 times as fast as our simulations. The program can be a valuable, real-time aid in the analysis and design of hypercube systems and techniques. In the next chapter we use the Path Analysis program to analyze the effectiveness of reroute methods.

CHAPTER 5.

RESULTS OF PATH ANALYSIS

Now we present results for performance degradation as determined from our Path Analysis program. In Table 2, we show the success rates of the different reroute strategies under various hypercube configurations. Since the Path Analysis program develops each path according to the FIND_PATHS algorithm, we can easily determine success and failure of the individual paths. The success rate of each reroute strategy is given for a number of different faulty conditions F_i of the hypercube. F_i contains i faulty links and/or nodes. Beneath each F_i we give the number of viable paths in that configuration. We call viable paths those paths which are from fault-free source to fault-free destination processors.

From this table, we can see that the table-routing methods [26], as expected, have the best success rate. Second best is *C&S adaptive* [27], but unreflected in this table is

Table 2. Success Rates for Various Reroute Strategies.

Reroute Strategy	F_2 (210)	F_3 (240)	F_4 (182)	F_5 (182)	F_6 (132)
none	178	194	125	111	86
C&S adaptive	210	240	182	175	132
hyperswitch	206	232	168	146	122
highfirst	205	228	166	147	112
reverse	206	226	162	146	110
table-centralized	210	240	182	182	132
table-distributed	210	240	182	182	132

the fact that *C&S adaptive* has a much greater average path length than the table-routing methods. *Hyperswitch* [11] is noticeably less successful than the above three, mostly due to its restriction to distance-length paths. Another item we notice is that *hyperswitch* is not overwhelmingly more successful than highfirst or reverse. As a fault-tolerance measure, *hyperswitch* may not be worth the extra expense. However, in this analysis we do not examine *hyperswitch*'s powerful congestion-avoiding capabilities. All the rerouting strategies show noticeable improvement in success over no reroute, giving evidence that rerouting is a valuable feature in a hypercube system. However, success data is only a part of Path Analysis. The real contribution of PA is its traffic and delay results.

We now present graphs of traffic as provided by Path Analysis for different faulty hypercubes. We can examine profiles of traffic on the links to evaluate the actual amount of traffic and the even distribution of traffic. In each figure we present traffic for the six different hypercube reroute strategies we have investigated. Figure 6 shows a node fault at 0000 with no task reconfiguration, Figure 7 shows a node fault at 0000 with modeled task reconfiguration, Figure 8 shows a link fault at -000, and Figure 9 shows the four-fault case we used in the previous chapter (F_4 above). All these graphs are for uniform destination distributions because these are more likely to bring out differences between reroute strategies.

These traffic profiles of the six reroute strategies show marked differences between them. However, we must remember when comparing traffic numbers that, for each link fault, *hyperswitch*, *highfirst*, and *reverse* automatically fail for two paths. The graphs of the four-fault case should be analyzed with column F_4 of the preceding Table 2 in mind.

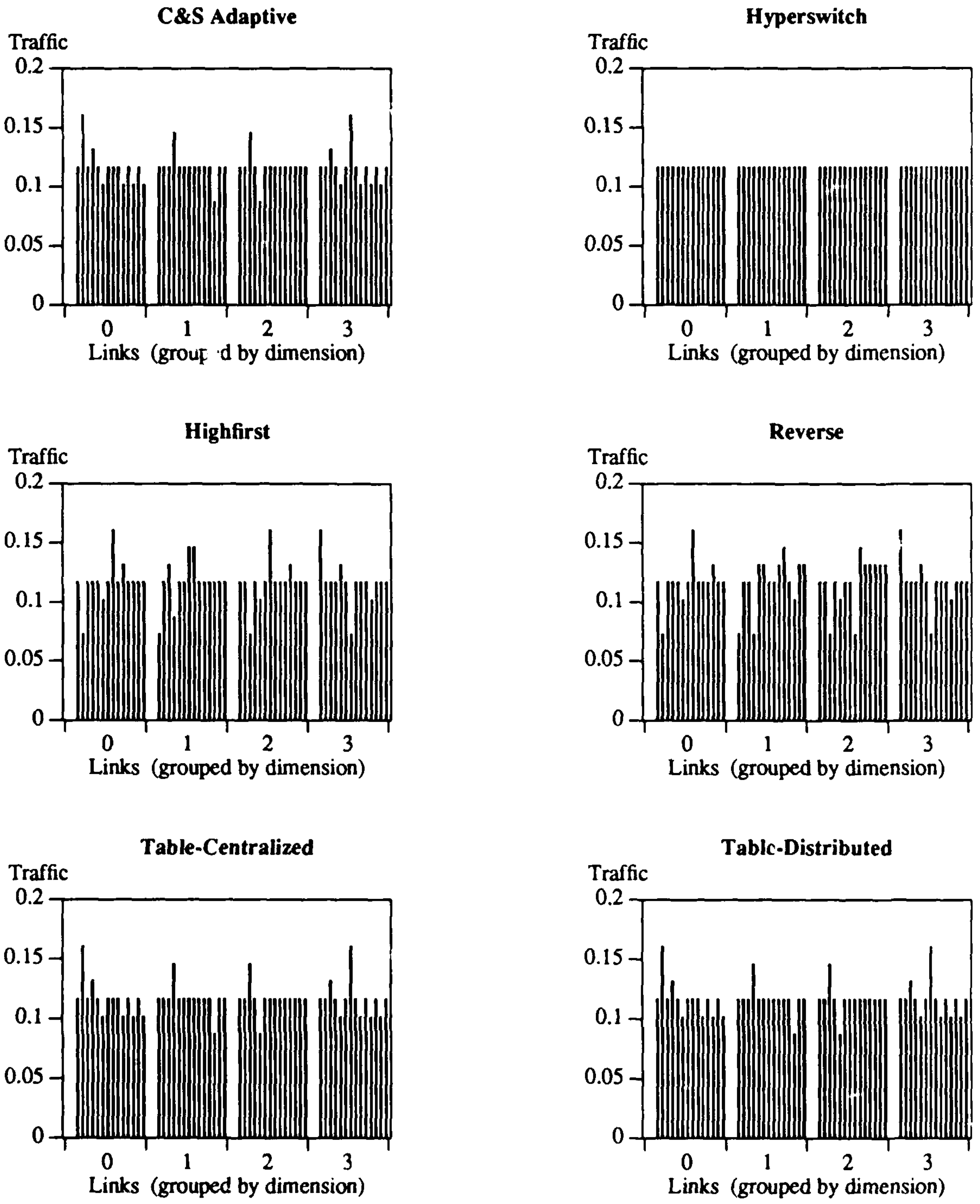


Figure 6. Path Analysis Results: Fault at Node 0000, Task Disappears

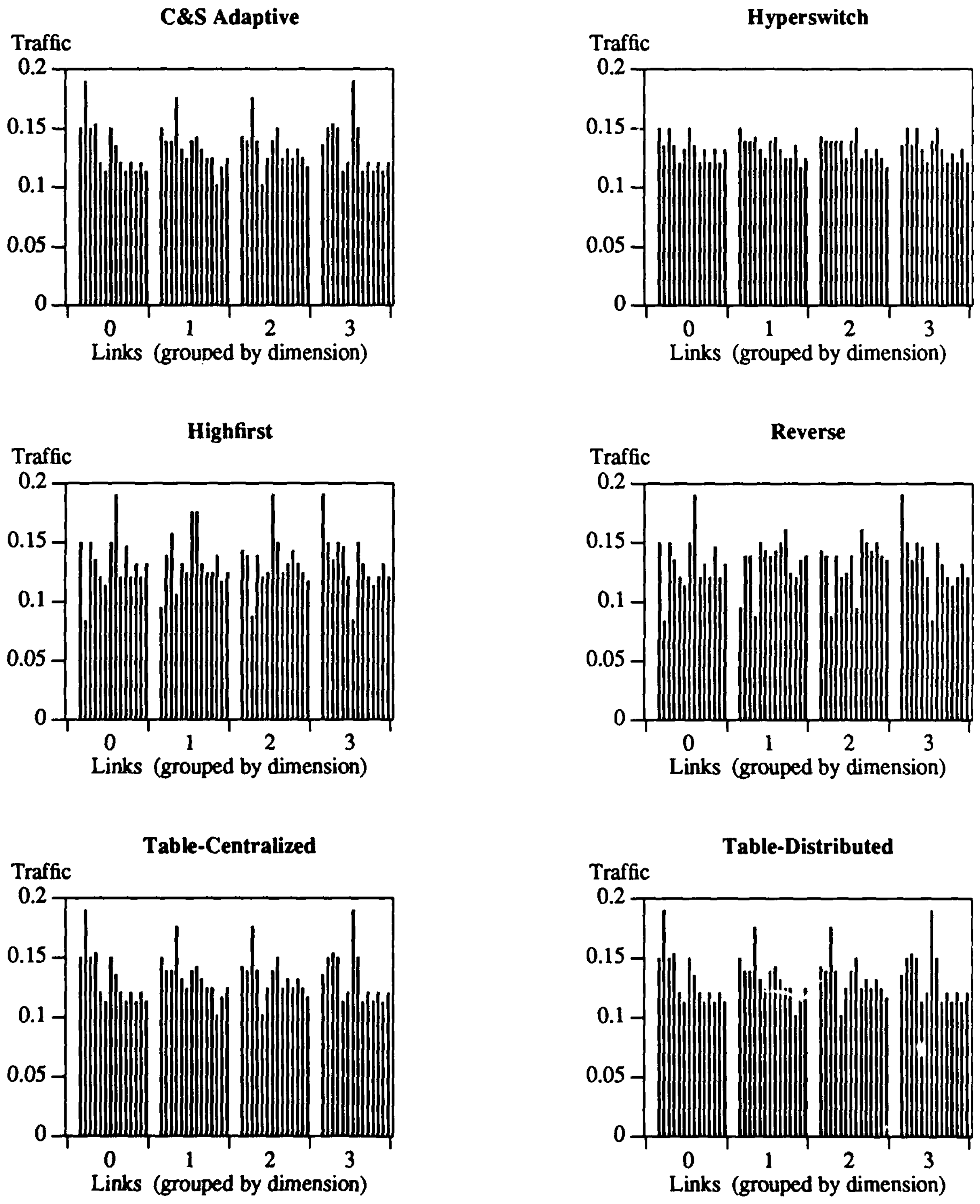


Figure 7. Path Analysis Results: Fault at Node 0000, Task Redistributed

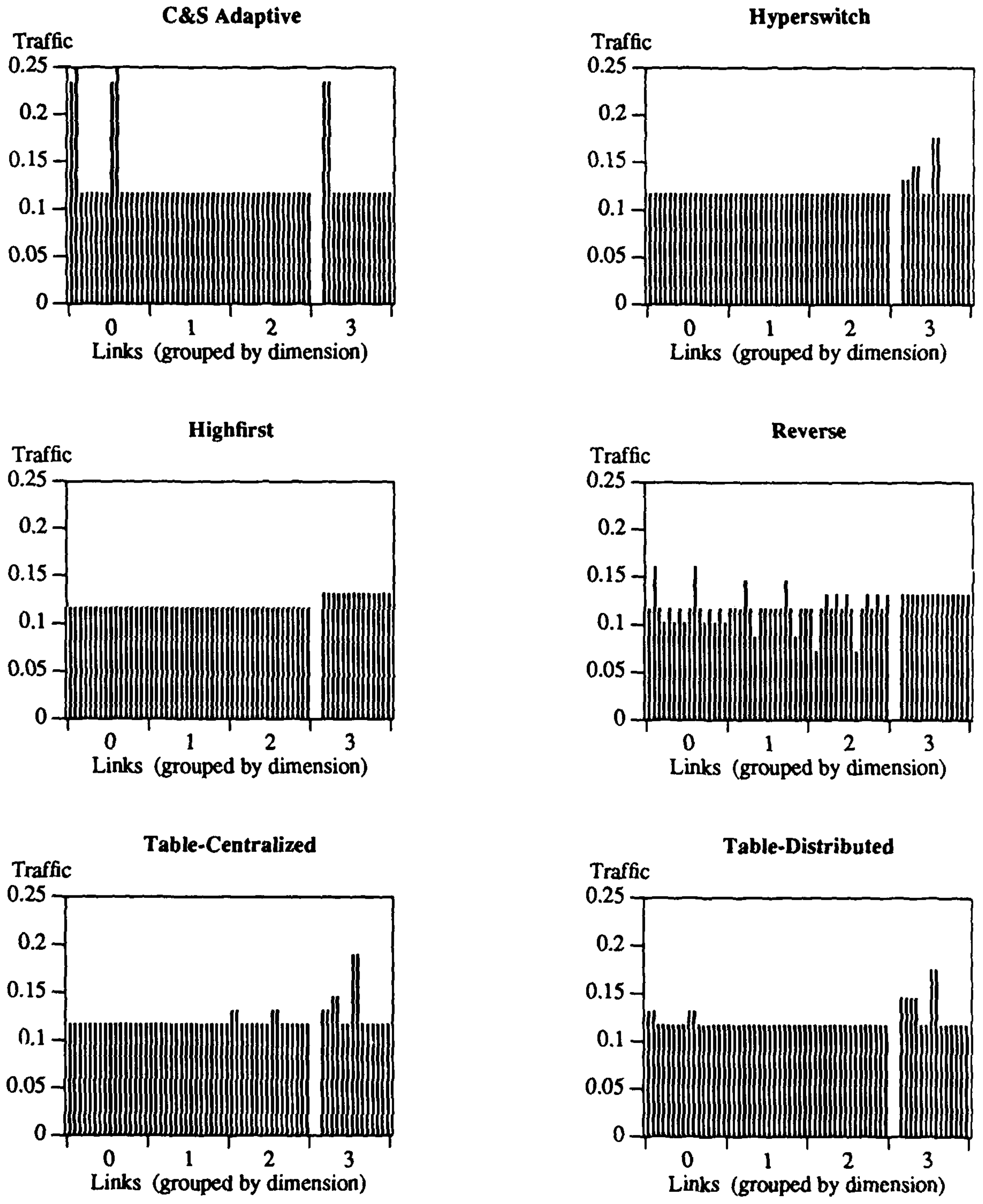


Figure 8. Path Analysis Results: Fault at Link -000

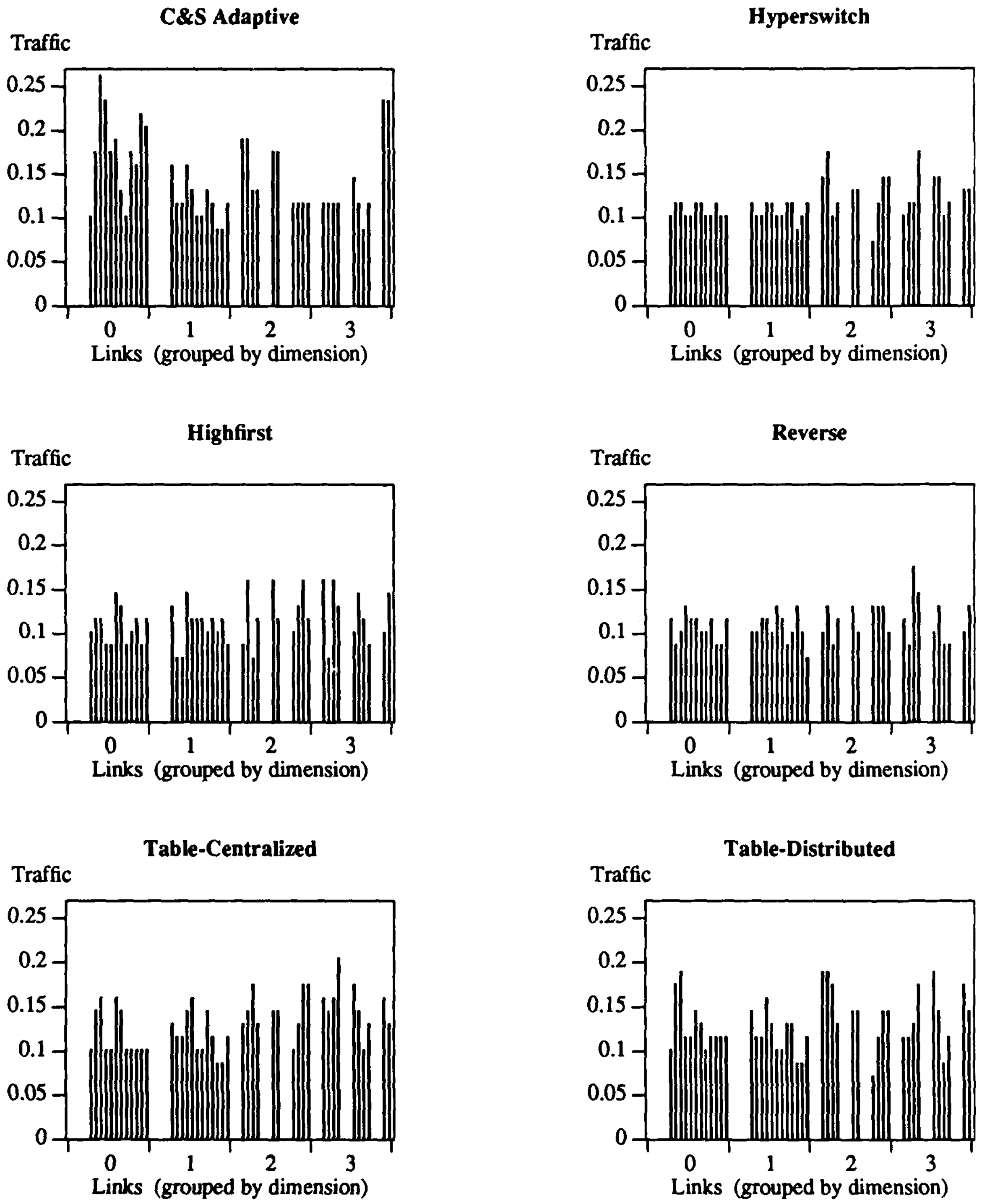


Figure 9. Path Analysis Results: Four-Fault Hypercube

There are interesting observations to be made from each of the figures. For instance, from Figure 6 it is clear that *hyperswitch* is best at equalizing the traffic across the links after a node fault has occurred. All the other strategies move the traffic lost from one link in a dimension to some other link in that dimension, leaving the distribution uneven.

Figure 7 demonstrates the greater unevenness in link traffic due to task redistribution. The basic quantitative effect of task redistribution under our model is evident in the difference between the *hyperswitch* graphs in this and the previous figures. The amount of added traffic strongly suggests that the modeling of task reconfiguration is an important factor in determining link traffic.

Figure 8 shows the differences among the reroute strategies when confronted with a link fault in the highest dimension. The regularity of *highfirst* is interesting. The *highfirst* traffic appears this way because sources evenly distributed throughout the hypercube know that their *ecube* paths will fail at link -000; each source compensates by taking another highest dimension link first, thus evenly distributing the traffic from -000 on these highest dimension links and not changing the traffic amounts on lower dimension links. We also note the large traffic amounts on certain links for *C&S adaptive*. This great unevenness is due to the fact that *C&S adaptive* has no lookahead; any message that has made its way to the third dimension link only to find it faulty must take a spare dimension, adding two to its path length.

Figure 9 shows the traffic in a four-fault hypercube, where the faults are at node 0000, node 0011, link 1-01, and link -110. Again recall that *hyperswitch*, *highfirst*, and *reverse* fail for a number of paths, and the traffic from these paths consequently does not appear on the graphs. Comparing the other three reroute strategies, we note that *C&S adaptive* gives greater total network traffic due to its greater average path length. The table-routing methods show less traffic than *C&S adaptive* because of their optimal length paths.

We can similarly produce graphs of delay from Path Analysis results. Together with success ratios and other incidental path information, traffic and delay can aid us in analyzing the condition of a faulty hypercube and the capabilities of various reroute strategies to handle it.

CHAPTER 6.

CONCLUSIONS

We have presented Path Analysis, a technique for calculating traffic on each of the many links of a hypercube. This analysis also provides delay in both packet-switched and the more current circuit-switched networks. Although a thorough comparison of reroute strategies is beyond the scope of this thesis, we have developed methods and tools which make such a comparison possible.

Path Analysis can also be extended to other network topologies with deterministic routing strategies. Arbitrary destination distributions are easily applied, and distributions from application program traces may be used. In fact, a program gathering paths with a variant of algorithm FIND_PATHS could give each $(s\ d)$ combination a precise traffic number derived from traces, ignoring the assumptions of uniform message sourcing and a destination distribution described by distance. Using extensions of the ideas in this thesis, the program could calculate link traffic and delay for these specific application traces.

REFERENCES

[1] T.-y. Feng, "A survey of interconnection networks," *IEEE Computer*, pp. 12-27, December 1981.

[2] D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Passing Parallel Processing*. Cambridge, MA: MIT Press, 1987.

[3] D. A. Rennels, "Fault tolerant computing - concepts and examples," *IEEE Transactions on Computers*, vol. C-33, pp. 1116-1129, Dec. 1984.

[4] J. P. Hayes, "A graph model for fault-tolerant computing systems," *IEEE Transactions on Computers*, vol. C-25, pp. 86-88, Jan. 1976.

[5] D. K. Pradhan, "Fault-tolerant multiprocessor link and bus network architectures," *IEEE Transactions on Computers*, vol. C-34, pp. 33-45, Jan. 1985.

[6] J. P. Hayes, T. N. Mudge, Q. T. Stout, S. Colley, and J. Palmer, "Architecture of a hypercube supercomputer," *Proc. 1986 Int. Conf. Parallel Processing*, pp. 653-660, Aug. 1986.

[7] Intel Scientific Computers, "iPSC: The First Family of Concurrent Supercomputers," 1985, product description.

[8] J. Tuazon, J. Peterson, M. Pniel, and D. Leberman, "Caltech/JPL Mark II hybercube concurrent processor," *Proc. 1985 Int. Conf. Parallel Processing*, pp. 666-673, Aug. 1985.

[9] J. C. Peterson, J. Tuazon, D. Lieberman, and M. Pniel, "The Mark III hypercube-ensemble concurrent computer," *Proc. 1985 Int. Conf. Parallel Processing*, pp. 71-73, Aug. 1985.

[10] S. F. Nugent, "The iPSC/2 direct-connect communications technology," *Proc. 3rd Conf. Hypercube Concurrent Computers and Applications*, Jan. 1988.

[11] E. Chow, H. Madan, J. Peterson, D. Grunwald, and D. Reed, "Hyperswitch network for the hypercube computer," *Proc. 15th Int. Symp. Computer Architecture*, pp. 90-99, May 1988.

[12] D. A. Rennels, "On implementing fault tolerance in binary hypercubes," *Proc. 16th Int. Symp. Fault-Tolerant Computing*, pp. 344-349, July 1986.

[13] S. Dutt and J. P. Hayes, "An automorphic approach to the design of fault-tolerant multiprocessors," *Proc. 19th Int. Symp. Fault-Tolerant Computing*, pp. 496-503, June 1989.

[14] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. S. Nair, K. Roy, and J. A. Abraham, "An evaluation of system-level fault tolerance on the Intel hypercube multiprocessor," *Proc. 18th Int. Symp. Fault-Tolerant Computing*, pp. 362-367, Jun. 1988.

[15] C. Aykanat and F. Ozguner, "A concurrent error detecting conjugate gradient algorithm on a hypercube multiprocessor," *Proc. 17th Int. Symp. Fault-Tolerant Computing*, pp. 204-209, July 1987.

[16] B. Becker and H. U. Simon, "How robust is the n-Cube?," *Proc. 27th Symp. Foundations Computer Science*, pp. 283-291, Oct. 1986.

[17] N. Graham, F. Harary, M. Livingston, and Q. Stout, "Subcube fault tolerance in hypercubes," Comp. Res. Lab. Tech. Report CRL-TR-12-87, University of Michigan, Sep. 1987.

[18] C. C. Li and W. K. Fuchs, "Graceful degradation on hypercube multiprocessors using data redistribution," *1989 Int. Conf. Parallel Processing*, Aug. 1989, (submitted).

[19] M. U. Uyar and A. P. Reeves, "Fault reconfiguration for the near neighbor problem in a distributed MIMD environment," *Proc. 5th Int. Conf. Distributed Computing Systems*, pp. 372-379, May 1985.

[20] S. H. Bokhari, "Partitioning problems in parallel, pipelined, and distributed computing," *IEEE Transactions on Computers*, vol. C-37, pp. 48-57, January 1988.

[21] G. Chuanshan, J. W. S. Liu, and M. Railey, "Load balancing algorithms in homogeneous distributed systems," *1984 Int. Conf. Parallel Processing*, pp. 302-306, Aug. 1984.

[22] A. K. Ezzat, R. D. Bergeron, and J. L. Pokoski, "Task allocation heuristics for distributed computing systems," *Proc. 6th Int. Conf. Distributed Computing Systems*, pp. 337-346, May 1986.

[23] V. B. Gylys and J. A. Edwards, "Optimal partitioning of workload for distributed systems," *Digest of Papers COMPCON*, pp. 353-357, Fall 1976.

[24] J. Hastad, T. Leighton, and M. Newman, "Reconfiguring a hypercube in the presence of faults," *Proc. 19th ACM Symp. Theory of Computing*, pp. 274-284, 1987.

[25] G. C. Fox and J. G. Koller, "A dynamic load balancer on the Intel hypercube," *Proc. 3rd Conf. Hypercube Concurrent Computers and Applications*, Jan. 1988.

[26] P. Banerjee, "Reconfiguring a hypercube in the presence of faults," *Proc. 4th Conf. Hypercube Concurrent Computers and Applications*, Mar. 1989.

[27] M. S. Chen and K. G. Shin, "Message routing in an injured hypercube," *Proc. 3rd Conf. Hypercube Concurrent Computers and Applications*, pp. 312-317, Jan. 1988.

[28] T. C. Lee and J. P. Hayes, "Routing and broadcasting in faulty hypercube computers," *Proc. 3rd Conf. Hypercube Concurrent Computers and Applications*, pp. 346-354, Jan. 1988.

[29] J. M. Gordon and Q. F. Stout, "Hypercube message routing in the presence of faults," *Proc. 3rd Conf. Hypercube Concurrent Computers and Applications*, pp. 318-327, Jan. 1988.

[30] H. Sullivan and T. R. Bashkow, "A large scale homogeneous fully distributed parallel machine," *Proc. 4th Symp. Computer Architecture*, pp. 105-117, Mar. 1977.

[31] L. Kleinrock, *Queueing Systems, Vol. II.* New York, NY: John Wiley and Sons, Inc., 1975.

[32] K. S. Trivedi, *Probability and Statistics with Reliability, Queueing and Computer Science Applications*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1982.

[33] H. D. Schwetman, "CSIM: A C-based, process-oriented simulation language," Tech. Report PP-080-85, Austin, Texas, 1985.